情報保護學會論文誌
第12券 第4號, 2002. 8

효율적인 다중 멱승 알고리즘과 그 응용

임 채 훈*

Efficient Multi-Exponentiation and Its Application

Chae Hoon Lim*

요 약

다수의 멱승의 곱을 계산하는 효율적인 다중 멱승 알고리즘은 다양한 암호 프로토콜의 성능 향상을 위해 유용하게 사용될 수 있다. 본 논문에서는 4가지의 서로 다른 다중 멱승 알고리즘을 기술하고 그 효율성을 비교 분석한다. 각 알고리즘은 곱해지는 멱승의 수에 따라 가장 효율적인 영역들이 있으며, 각 영역에서 최선의 알고리즘을 이용하는 경우 기본적인 이진 알고리즘을 사용하는 경우에 비해 2~4배 정도의 성능향상을 얻을 수 있고, 각각의 멱승을 독립적으로 계산하는 경우에 비해서는 2~10배 정도의 성능향상을 얻을 수 있다.

ABSTRACT

This paper deals with efficient algorithms for computing a product of n distinct powers in a group(called multi-exponentiation). Four different algorithms are presented and analyzed, each of which has its own range of n for best performance. Using the best performing algorithm for n ranging from 2 to several thousands, one can achieve 2 to 4 times speed-up compared to the baseline binary algorithm and 2 to 10 times speed-up compared to individual exponentiation.

Keyword : *Multi-exponentiation, Digital signatures, Batch verification*

I. Introduction

Evaluating exponentiation in a group is the most time-consuming operation in most public key cryptosystems, e.g., modular exponentiation in the RSA and Diffie-Hellman/ElGamal systems and scalar multiplication in elliptic curve cryptosystems. Fast exponentiation algorithms thus have been studied extensively in past years and a number of such algorithms have been proposed. For example, the sliding window algorithm is the most popular algorithm for general exponentiation in any group[1,2] and there exist much faster algorithms for exponentiation to a fixed base [3,4].

On the other hand, it is often required in many cryptographic protocols to evaluate a product of multiple powers in a group (called multi-exponentiation). Verification of discrete log-based signatures is the most common example : Individual verification requires two-term exponentiation and batch verification [5,6] requires n-term exponentiation for quite large n. There exist several efficient algorithms for two-term exponentiation [7,8], but not much research has been done for efficient algorithms for general multi-exponenti-

* 세종대학교 인터넷학과(chlim@sejong.ac.kr)

ation with large n.

In this paper, we present and analyze several efficient algorithms for n-term exponentiation for arbitrary large n. The presented algorithms include the sliding window algorithms with/without signed encoding and algorithms using the precomputation techniques of [3] and [4]. Each of the algorithms is shown to have its own range of n with relative strength. We also show that with signed encoding and Montgomery's simultaneous inversion technique [9, Algorithm 10.3.4], these algorithms can be significantly improved in elliptic curve groups. Finally, we demonstrate the power of multi-exponentiation with application to batch verification of (modified) DSA signatures.

II. Batch Verification of Exponentiation

Let G be a (cyclic) group of order $q=|G|$ and g_i $(0 \le i < n)$ be elements of order q in G. Given a collection of triplets (x_i, g_i, y_i) $(0 \le i < n)$ such that $x_i \in Z_q$ and $g_i, y_i \in G$, we want to verify that each triplet (x_i, g_i, y_i) satisfies the exponentiation function $y_i = g_i^{x_i}$ in G.

A naive approach to doing such verification is to evaluate each exponentiation $g_i^{x_i}$ and compare the result with y_i, which requires n exponentiations. However, there is a much better approach: the probabilistic batch verification [5] (called the small exponent test in [6]). In this probabilistic batch verification, we randomly pick integers c_i $(0 \le i < n)$ over the interval $[0, 2^t)$ and test the following equation for equality:

$$\prod_{i=0}^{n-1} y_i^{c_i} = \prod_{i=0}^{n-1} g_i^{z_i}, \qquad (1)$$

where $z_i = c_i x_i \mod q$. The probability of error in this probabilistic batch verification is shown to be at most 2^{-t} [6], so we can obtain a desired level of confidence by choosing a appropriate value of t. In most applications, t=30~60 would suffice.

If the exponentiation function has the same base g(i.e., $g_i = g$ for all i's), the right-hand side of Equation (1) is simplified to a single power g^z, where $z = \sum c_i x_i \mod q$. In this case, the Bucket Test in [6] can further speed up the above small exponent test, but it cannot be used for batch verification of general exponentiation with distinct bases.

It can thus be seen that in either case efficient multi-exponentiation is a key to the performance of batch verification of modular exponentiation. The goal of this paper is to develop such algorithms. From now on, we will focus on efficient evaluation of the following form of general multi-exponentiation:

$$Y = \prod_{i=0}^{n-1} y_i^{c_i}, \text{ where } c_i \in [0, 2^t) \qquad (2)$$

III. Algorithms for Multi-Exponentiation

In this section, we will present six algorithms for multi-exponentiation. We first describe the basic binary algorithms that can be used as a baseline for performance comparison. Then four algorithms are presented to speed up the naive binary algorithms: sliding window algorithms using unsigned/signed encoding, and algorithms using the precomputation techniques of Brickell et al. [3] and Lim-Lee [4]

Throughout this paper, we will use the letters M, S and I to denote the computing complexity of multiplication, squaring and inversion, respectively, and the letters γ and μ to denote the complexity ratio of S over M and I over M, respectively, i.e., $\gamma = S/M$, $\mu = I/M$.

3.1 Binary Algorithms

Let $c_i = \sum_{i=0}^{t-1} c_{i,j} 2^j$ be the binary representation of c_i, where $c_{i,j} \in \{0,1\}$. The binary algorithm to compute Y in Equation (2) carries out the following iteration t times, starting with $Y=1$ and running $j=t-1$ down to 0: multiply Y by all y_i's such that $c_{i,j}=1$ and then square the result Y. Obviously, this algorithm requires $0.5nt-1$ multiplications and $t-1$ squarings on average. With optimal signed encoding for each exponent(e.g., see [10]), the expected number of multiplications can be reduced to $\frac{n(t+1)}{3}-1$ at the cost of t more squarings. This performance can be obtained by computing Y as

$$Y = \left(\prod_{i=0}^{n-1} y_i^{c_i^p}\right) \cdot \left(\prod_{i=0}^{n-1} y_i^{c_i^m}\right)^{-1},$$

where c_i^p and c_i^m respectively denote the positive and negative parts of c_i after optimal signed encoding(so, $c_i = c_i^p - c_i^m$). Note that optimal signed encoding can reduce the probability of a bit being nonzero to 1/3 on average with at most 1- bit expansion.

The following theorem summarizes the performances of the unsigned binary algorithm(denoted by Algorithm **BU**) and the signed binary algorithm(denoted by Algorithm BS) for computing n-term exponentiation.

Theorem 1 (Algorithms **BU**/**BS**)

a) The expected number of multiplications required by Algorithm **BU** is given by

$$C_{BU}(n;t) = \frac{nt}{2} - 1 + \gamma(t-1).$$

b) The expected number of multiplications required by Algorithm **BS** is given by

$$C_{BS}(n;t) = \frac{n(t+1)}{3} - 1 + 2\gamma t + \mu.$$

3.2 Sliding Window Algorithms

The window-family algorithm will be a better way to evaluate n-term exponentiation with small n ($n=2$ to 4). Among many variants, we focus on the window algorithm using independently sliding windows[8], since it is most efficient for application to multi-exponentiation. This window algorithm uses a distinct sliding window for each exponent instead of an ordinary simultaneous window.

Let w be the window size in bits and k_i be the number of windowed values for c_i. Then we can express each exponent c_i as

$$c_i = \sum_{j=0}^{k_i-1} c_{i,j} 2^{l_{i,j}},$$

where $c_{i,j}$'s are odd and $0 < c_{i,j} < 2^w$. Let $Y_{i,j}$'s be the precomputed values for y_i given by

$$Y_{i,j} = y_i^{2j-1} \text{ for } 1 \le j \le 2^{w-1}.$$

Then, Equation (2) can be rewriten as

$$Y = \prod_{i=0}^{n-1} y_i^{c_i} = \prod_{i=0}^{n-1}\left(\prod_{j=0}^{k_i-1} y_i^{c_{i,j} 2^{l_{i,j}}}\right) = \prod_{i=0}^{n-1}\left(\prod_{j=0}^{k_i-1} Y_{i,(c_{i,j}+1)/2}^{2^{l_{i,j}}}\right) \tag{3}$$

Now the right-hand side of Equation (3) can be computed using the ordinary square-and-multiply algorithm as shown in Algorithm **WU**.

Algorithm WU: Sliding window Alg.

```
INPUT: $y_i$, $c_i$ $(0 \le i < n)$, $G$
OUTPUT: $Y = \prod_{i=0}^{n-1} y_i^{c_i}$ in $G$
  0: encoding: $c_i = (c_{i,j}, l_{i,j})$
  1: for $i=0$ to $n-1$ step $+1$
  2:     $Y_{i,1} \leftarrow y_i$, $T \leftarrow y_i^2$
  3:     $Y_{i,j} \leftarrow Y_{i,j-1} \cdot T$ $(2 \le j \le 2^{w-1})$
  4: $pos \leftarrow \max\{l_{i,k_i}\}_{i=0}^{n-1}$
  5: $Y \leftarrow \prod_{l_{i,k_i}=pos} Y_{i,(c_{i,k_i}+1)/2}$
```

```
6: while ( $pos > 0$)
7:     $pos \leftarrow pos - 1$
8:     $Y \leftarrow Y^2$
9:     $Y \leftarrow Y \cdot \prod_{l_{i,j}=pos} Y_{i,(c_{i,j}+1)/2}$
10: return $Y$
```

The precomputation stage (steps 1 to 3) requires $(2^{w-1}-1)$ multiplications and 1 squaring for each exponent, and the main computation part (steps 4 to 9) requires $(t-1)$ squarings and about $\frac{t}{w+1}$ multiplications for each exponent. Theorem 2 below summarizes the performance of Algorithm **WU**:

Theorem 2 (Algorithm **WU**)

a) The expected number of multiplications required by Algorithm **WU** with window size w is given by

$$C_{WU}(n;t,w) = \left(\frac{t}{w+1} + 2^{w-1} - 1 + \gamma\right)n + \gamma(t-1) - 1.$$

The algorithm also requires a temporary storage for $2^{w-1}n$ precomputed values.

b) The optimal window size w_{opt} only depends on t. The range of t for which w_{opt} is optimal can be determined from the inequality $C_{WU}(n;t,w_{opt}) \le C_{WU}(n;t,w_{opt}+1)$:

$$t \le 2^{w_{opt}-1}(w_{opt}+1)(w_{opt}+2).$$

This inequality gives the following table for optimal window sizes depending on the range of t, where each pair ($t_{\max}$, w_{opt}) means that w_{opt} is optimal up to $t_{\max}$ (from the previous value):

w_{opt}	2	3	4	5	6
$t_{\max}$	24	80	240	672	1792

We next consider how to use signed encoding in the above sliding window algorithm. Since signed encoding requires expensive multiplicative inversion, we want to minimize the number of inverses required. For this, we split each sign-encoded exponent into positive and negative parts as in Algorithm **BS**. Suppose that each exponent c_i is sign-encoded with window size w. We can then express each c_i as

$$c_i = \sum_{j=0}^{k_i-1} c_{i,j} 2^{l_{i,j}} = \sum_{j \in k_i^p} c_{i,j}^p 2^{l_{i,j}} - \sum_{j \in k_i^m} c_{i,j}^m 2^{l_{i,j}},$$

where $c_{i,j}$'s are odd and $0 < |c_{i,j}| < 2^{w-1}$, and k_i^p and k_i^m respectively denote the set of indices for positive and negative windowed values. Equation (2) can then be expressed as

$$Y = \prod_{i=0}^{n-1} y_i^{c_i} = \prod_{i=0}^{n-1}\left(\prod_{j \in k_i^p} y_i^{c_{i,j}^p 2^{l_{i,j}}}\right) \cdot \prod_{i=0}^{n-1}\left(\prod_{j \in k_i^m} y_i^{c_{i,j}^m 2^{l_{i,j}}}\right)^{-1} \qquad (4)$$

This equation enables us to use signed encoding in the sliding window algorithm at the cost of one inversion and at most (t-w) squarings. Let **WS** be the algorithm to compute the right-hand side of equation (4). Note that the number of nonzero windows in each sign-encoded exponent is about $\frac{t+1}{w+2}$ on average. Therefore, Algorithm **WS** will perform better than Algorithm **WU** only if the cost reduction (i.e., the the number of multiplications reduced) due to the signed encoding is greater than the increased cost of one inversion and $(t-w)$ squarings. This would be the case for large n. The following theorem summarizes the performance of Algorithm **WS**:

Theorem 3 (Algorithm **WS**)

a) The expected number of multiplications required by Algorithm **WS** with window size w is given by

$$C_{WS}(n;t,w)=\left(\frac{t+1}{w+2}+2^{w-1}+\gamma-1\right)n + \gamma(2t-w)+\mu-1.$$

The algorithm also requires a temporary storage for $2^{w-1}n$precomputed values.

b) The optimal window size w_{opt} mainly depends on t. Given n, the range of t for which w_{opt} is optimal can be determined by solving $C_{WS}(n;t,w_{opt})\le C_{WS}(n;t,w_{opt}+1)$:

$$t\le\left(2^{w_{opt}-1}-\frac{\gamma}{n}\right)(w_{opt}+2)(w_{opt}+3)-1 \approx 2^{w_{opt}-1}(w_{opt}+2)(w_{opt}+3)-1$$

where the latter approximation holds for $n\gg(w_{opt})^2$ (not that we always have $\gamma<1$). This inequality gives the following table for optimal window sizes for an interesting range of t:

w_{opt}	2	3	4	5	6
$t_{\max}$	39	119	335	895	2303

3.3 Algorithm Using Lim-Lee's Precomputation Technique

It is quite natural to come up with Lim-Lee's precomputation technique [4] for efficient n-term exponentiation(note that this technique is a natural extension of, so including, the simultaneous window algorithm, often called Shamir's trick for $n=2$). Basic idea of Lim-Lee's technique with parameters (h,w) is to partition the set of n powers into h blocks of size w, make a distinct precomputation table for simultaneous window of size 1 for each block, and then apply the binary algorithm to the h blocks of powers.

More formally, let $c_i=\sum_{i=0}^{t-1}c_{i,j}2^j$ be the binary representation of c_i and express Equation (2) as

$$Y=\prod_{i=0}^{n-1}y_i^{c_i}=\prod_{j=0}^{t-1}\left(\prod_{i=0}^{n-1}y_i^{c_{i,j}}\right)^{2^j} = \prod_{j=0}^{t-1}\left(\prod_{k=0}^{h-1}\left(\prod_{i=kw}^{(k+1)w-1}y_i^{c_{i,j}}\right)\right)^{2^j}, \qquad (5)$$

where $h=\lceil\frac{n}{w}\rceil$. Next, precompute and store the products of all possible combinations of y_i's in each k-th block of size w as

$$Y_{k,e}=\prod_{j=0}^{w-1}y_{kw+j}^{e_j},$$

where $0\le k<h$ and $0\le e=e_{w-1}\cdots e_1e_0<2^w$. Then the right-hand side of equation (5) can be computed as

$$Y=\prod_{j=0}^{t-1}\left(\prod_{k=0}^{h-1}Y_{k,e(k)}\right)^{2^j},$$

where $e(k)=c_{(k+1)w-1,j}\cdots c_{kw+1,j}c_{kw,j}$. The detailed algorithm is depicted below as Algorithm **LL**.

```
Algorithm LL: Lim-Lee's algorithm
  INPUT: $y_i, c_i\ (0\le i<n)$, $G$
  OUTPUT: $Y=\prod_{i=0}^{n-1}y_i^{c_i}$ in $G$
    1: for $k=0$ to $h-1$ step +1
    2:     for $e=0$ to $2^w-1$ step +1
    3:         $e=\sum_{i=0}^{w-1}e_i2^i$,  $Y_{k,e}\leftarrow\prod_{i=0}^{w-1}y_{kw+i}^{e_i}$
    4: $Y\leftarrow 1$
    5: for $k=0$ to $h-1$ step +1
    6:     $e\leftarrow\sum_{i=kw}^{(k+1)w-1}c_{i,t-1}2^{i-kw}$
    7:     $Y\leftarrow Y\cdot Y_{k,e}$
    8: for $j=t-2$ to 0 step -1
    9:     $Y\leftarrow Y^2$
   10:     for $k=0$ to $h-1$ step +1
```

```
11:         $e \leftarrow \sum_{i=kw}^{(k+1)w-1} c_{i,j} 2^{i-kw}$
12:         $Y \leftarrow Y \cdot Y_{k,e}$
13: return $Y$
```

It is easy to see that the precomputation stage (steps 1 to 3) requires $(2^w-w-1)h$ multiplications and the main computation (steps 4 to 12) can be done in $(t-1)$ squarings and at most $(th-1)$ multiplications. For the average performance, we need to consider the expected number of all-zero e's (see [4] for details). The following theorem summarizes the performance of Algorithm **LL**:

Theorem 4 (Algorithm LL)

a) The expected number of multiplications required by Algorithm **LL** with blocking factor w is given by

$$C_{LL}(n;t,w)=\left(\frac{2^w-1}{2^w}t+2^w-w-1\right)h + \frac{2^r-1}{2^r}t+2^r-r-2+\gamma(t-1),$$

where $h=\lceil \frac{n}{w} \rceil$ and $r=n \bmod w$ if $n>w$, and $w=n$ (so $h=1$, $r=0$) if $n\le w$. The algorithm also requires a temporary storage for $2^w\lceil \frac{n}{w}\rceil$ precomputed values.

b) The optimal value of blocking factor w, w_{opt}, mainly depends on t. Given n, the range of t for which w_{opt} is optimal can be determined from the inequality $C_{LL}(n;t,w_{opt})\le CC_{LL}(n;t,w_{opt}+1)$. In particular, when n is a multiple of both w_{opt} and $w_{opt}+1$, one can obtain the following simple formula:

$$t\le\frac{1+(w_{opt}-1)2^{w_{opt}}}{1-(w_{opt}+2)2^{-(w_{opt}+1)}}$$

This inequality gives the following table for optimal window sizes for an interesting range of t:

w_{opt}	2	3	4	5	6	7	8
$t_{\max}$	10	24	60	144	342	797	1828

Note that the average performance of Algorithm **LL** is slightly worse when n is not a multiple of w. So, it is always preferable to choose the batch size n as a multiple of w_{opt} whenever possible.

3.4 Algorithm Using Brickell et al.'s Precomputation Technique

Another way to speed up n-term exponentiation (in particular, for large n) is to use the basic scheme of Brickell et al.'s precomputation method [3]. Suppose that for a fixed window size w each exponent c_i is represented in base 2^w *as* $c_i=\sum_{j=0}^{h-1} c_{i,j}2^{jw}$, where $h=\lceil \frac{t}{w}\rceil$ and $0\le c_{i,j}<2^w$. Then we can express Equation (2) as

$$Y=\prod_{i=0}^{n-1} y_i^{c_i}=\prod_{i=0}^{n-1}\left(\prod_{j=0}^{h-1} y_i^{c_{i,j}2^{jw}}\right)=\prod_{j=0}^{h-1} Y_j^{2^{jw}}, \qquad (6)$$

where $Y_j=\prod_{i=0}^{n-1} y_i^{c_{i,j}}$. Now we can compute each Y_j using the basic scheme in [3] and the right-hand side of Equation (6) using the repeated square-and-multiply algorithm. See Algorithm **BG** for details.

Algorithm BG: Brickell et al.'s algorithm

INPUT: y_i, c_i $(0\le i<n)$, G

OUTPUT: $Y=\prod_{i=0}^{n-1} y_i^{c_i}$ in G

```
1: $Y \leftarrow 1$
2: for $j=h-1$ to 0 step -1
3:     $T \leftarrow Z \leftarrow \prod_{c_{i,j}=2^w-1} y_i$
4:     for $k=2^w-2$ to 1 step -1
5:         $T \leftarrow T \cdot \prod_{c_{i,j}=k} y_i$
```

```
 6:        $Z \leftarrow Z \cdot T$
 7:    $Y \leftarrow Y \cdot Z$
 8:    for $k=0$ to $w-1$ step $+1$
 9:        $Y \leftarrow Y^2$
10: return $Y$
```

Note that each c_i is grouped by w bits from the LSB for simplicity, it would be better to do the grouping starting with the MSB, since then the number of squarings can be a little bit reduced when $t \neq 0$ mod w. The following theorem summarizes the performance of Algorithm **BG**:

Theorem 5 (Algorithm BG)

a) The expected number of multiplications required by Algorithm **BG** with base 2^w is given by

$$\begin{aligned} C_{BG}(n;t,w) = &\left(\frac{2^w-1}{2^w}n + 2^w - 2\right)h \\ &+ \left(\frac{2^r-1}{2^r} + 2^r - 2\right)\delta(r) \\ &+ \gamma(t-w) - 1, \end{aligned}$$

where $h = \lfloor \frac{t}{w} \rfloor$, $r = t$ mod w, and $\delta(r) = 1$ if $r \neq 0$ and $\delta(r) = 0$ otherwise.

b) The optimal window size w_{opt} mainly depends on n. Given t, the range of n for which w_{opt} is optimal can be determined by $C_{BG}(n;t,w_{opt}) \leq C_{BG}(n;t,w_{opt}+1)$.In particular, when t is a multiple of both w_{opt} and $w_{opt}+1$, one can obtain the following simple inequality:

$$\begin{aligned} n &\leq \frac{(2+(w_{opt}-1)2^{w_{opt}})t - \gamma w_{opt}(w_{opt}+1)}{1-(w_{opt}+2)2^{-(w_{opt}+1)}t} \\ &\approx \frac{2+(w_{opt}-1)2^{w_{opt}}}{1-(w_{opt}+2)2^{-(w_{opt}+1)}}, \end{aligned}$$

where the latter approximation holds for $t \gg (w_{opt})^2$.

The average performance of Algorithm BG is slightly worse when t is not a multiple of w. Note however that the bit-length t of exponent is a security parameter of a system and thus cannot be chosen arbitrarily. Thus the optimal size of w should be determined for a given t. For example, one can obtain the following table for $t=160$.

w_{opt}	2	3	4	5	6
n_{max}	11	25	61	148	324

w_{opt}	7	8	9	10	11
n_{max}	776	1892	3826	12269	23513

We can obtain a more general version of Algorithm BG using some on-line precomputation. For example, suppose that the n vlaues, $z_i = y_i^{2^w}$ for $0 \leq i < n$, are on-line precomputed. Then, Equation (6) can be rewritten as

$$\begin{aligned} Y &= \prod_{j=0,j+=2}^{h-1} \left(\prod_{i=0}^{n-1} y_i^{c_{i,j}} z_i^{c_{i,j+1}} \right)^{2^{jw}} \\ &= \prod_{j=0,j+=2}^{h-1} (Y_j Z_j)^{2^{jw}}, \qquad (7) \end{aligned}$$

where $Y_j = \prod_{i=0}^{n-1} y_i^{c_{i,j}}$ and $Z_j = \prod_{i=0}^{n-1} y_i^{c_{i,j+1}}$. Note that using Equation (7) one can reduce the number of multiplications required in step 6 almost by half at the cost of wn squarings for on-line precomputation. Therefore, Using this equation may result in better performances when t is large and n is relatively small.

In general, Algorithm **BG** using v pre- for $1 \leq j \leq v$) has the following performance formula:

$$\begin{aligned} C_{BG}(n;t,w,v) = &\left(\frac{2^w-1}{2^w}h + \frac{2^r-1}{2^r}\right)n \\ &+ h'(2^w-2) + (2^{w'}-2)\delta(w') \\ &+ \gamma(t-w+vw(n-1)) - 1, \end{aligned}$$

where $0 \leq v < h$, $h = \lfloor \frac{t}{w} \rfloor$, $r = t$ mod w, $h' = \lfloor \frac{h}{v+1} \rfloor$ and $w' = r$ if $h = 0$ mod $v+1$, $w' = w$ otherwise. Note that $C_{BG}(n;t,w,0)$

[Table 1] Range of n giving best performances in Algorithm BG for given v and t ($a\sim b$ denotes $a\leq n\leq b$)

t \ v_{opt}	≥4	3	2	1	0
60	2~3	none	4~8	~31	32~
160	2~9	~18	~38	~168	169~
256	2~14	~30	~91	~405	406~
512	2~65	~91	~270	~1984	1985~
768	2~78	~318	~635	~4855	4856~
1024	2~239	~437	~1346	~10696	10697~

$= C_{BG}(n; t, w)$.

To achieve the best possible performance for given n and t, we have to choose optimal values for w and v. The value of v, which determines the amount of on-line precomputation, may be quite large for small n and large t. [Table 1] shows the range of n according to the optimal values of v for some interested values of t. For example, when $t=160$, using nonzero v is always advantageous if $n\leq 168$. Obviously, the performance advantage with on-line precomputation becomes larger as t increases.

Ⅳ. Performance Comparison

Let us compare the performances of Algorithms **WU**, **WS**, **LL** and **BG**. First note that the optimal window size w_{opt} in Algorithms **WU** and **WS** only depends on the bit-length t of exponents and the optimal blocking factor w_{opt} in Algorithm **LL** depends only on t assuming that $n=0$ mod w_{opt}. Thus, in these three algorithms, for given t we can express the cost function in terms of n alone. For example, for $t=160$, we have the following simple formulas:

$$C_{WU}(n; 160, 4) = 39.8n + 126.2,$$

$$C_{WS}(n; 160, 4) = 34.63n + 251.8 + \mu,$$

$$LL(n; 160, 6) = \begin{cases} 160(1-2^{-n}) \\ +2^{n} - n + 125.2, \text{ if } n<6 \\ 35.74n + 126.2 \text{ if } n=0 \text{ mod } 6. \end{cases}$$

Here we took $\gamma = S/M = 0.8$. Note that if our objective is to verify the equality $Y = A \cdot B^{-1}$, then we can check the equality by $Y \cdot B = A$ without computing the multiplicative inverse. So, for simplicity, let us take $\mu = I/M = 0$ in Algorithm **WS** and denote the resulting algorithm as Algorithm **WS***.

On the other hand, the optimal window size in Algorithm **BG** depends on both n and t. For example, for $t=160$, we can obtain the following optimal performance formulas according to the range of n:

$$C_{BG}(n; 160, 6, 1) = 31.33n + 937.4 (93 \leq n \leq 168),$$
$$C_{BG}(n; 160, 6, 0) = 26.53n + 1748 (169 \leq n \leq 324),$$
$$C_{BG}(n; 160, 7, 0) = 22.81n + 2955 (325 \leq n \leq 776),$$
$$C_{BG}(n; 160, 8, 0) = 19.92n + 5200 (777 \leq n \leq 1893),$$
$$C_{BG}(n; 160, 9, 0) = 17.96n + 8916 (1894 \leq n \leq 3825),$$
$$C_{BG}(n; 160, 10, 0) = 15.98n + 16470 (3826 \leq n \leq 12270)$$

Using the above formulas derived for $t=160$, we can determine the best algorithm according to the batch size n. We can derive similar equations for other values of t. [Table 2] shows the best performing algorithms and their range of n for some selected values of t. Note that we almost always have the same order of preferred algorithms as n increases: **WU**, **LL**, **WS***, **BG** (in fact, there are some fluctuations in the order of Algorithm **LL** and **WS***, though neglected in the table).

[Table 2] Best performing algorithms for given n and t

Best Alg.	WU	LL	WS*	BG
$t=60$	$2\leq n<4$	$4\leq n<60$	$160\leq n<72$	$72\leq n$
$t=160$	$2\leq n<5$	$5\leq n<120$	$120\leq n<186$	$186\leq n$
$t=256$	$2\leq n<5$	$5\leq n<240$	$240\leq n<252$	$252\leq n$
$t=512$	$2\leq n<6$	$6\leq n<420$	none	$420\leq n$
$t=768$	$2\leq n<6$	$6\leq n<456$	none	$456\leq n$
$t=1024$	$2\leq n<6$	$6\leq n<608$	none	$608\leq n$

[Table 3] No. of multiplications(unit : 1000) required for multi-exponentiation for $t=160$

n	BU	WU	WS*	LL	BG	BU/ Best
2	0.29	0.21	0.32	0.25	0.27	1.38
5	0.53	0.33	0.43	**0.31**	0.46	1.71
30	2.53	1.32	1.29	**1.20**	1.62	2.11
60	4.93	2.51	2.33	**2.27**	2.70	2.17
90	7.33	3.71	3.37	**3.34**	3.75	2.19
120	9.73	4.90	**4.41**	4.42	4.70	2.21
150	12.13	6.10	5.45	5.49	5.64	2.23
300	24.13	12.07	10.64	10.85	**9.71**	2.49
600	48.13	24.01	21.03	21.58	**16.64**	2.89
1.2K	96.13	47.89	41.81	40.03	**29.11**	3.30
3K	240.13	119.53	104.15	107.38	**62.79**	3.82
6K	480.13	238.93	208.05	214.63	**112.37**	4.27

For more exact quantitative comparison, we provide performance figures of the four algorithms in [Table 3] for some selected values of n, where the performance of the binary algorithm is also tabulated as it can serve as a base line for the comparison. The table shows that with the best performing algorithm for given n we can achieve about 2 to 4 times speed-up(for the range of n from several tens to several thousands) compared to the binary algorithm.

V. Further Speedup in Elliptic Curve Groups

Basic operation in elliptic curve arithmetic is addition and subtraction between elliptic points, and computation of an integer multiple of a given point in this additive group, called scalar multiplication, corresponds to modular exponentiation in multiplicative groups. So, multi-exponentiation of Equation (2) can be written using additive notation as

$$Y=\sum_{i=0}^{n-1} c_i Y_i,$$

where Y_i's are base points and c_i's are scalar values(integers). A distinct feature of elliptic curve arithmetic, compared to modular arithmetic, is that we can freely use addition/subtraction chains since the cost for elliptic subtraction is almost the same as the cost for elliptic addition. This enables us to improve the algorithms presented in Sect.III when used for elliptic curve arithmetic. Furthermore, we can take advantage of the high degree of parallelism in Algorithms **LL** and **BG** to further improve these algorithms.

In this section, we will use γ to denote the performance ratio of elliptic doubling-to-addition, i.e., $\gamma=D/A$.

4.1 Improvement Using Signed Encoding

A basic algorithm for scalar multiplication in elliptic curve groups is the signed binary algorithm, denoted by Algorithm **BS-EC**. From Sect.III.1, we can easily see that the expected number of elliptic additions required by Algorithm **BS-EC** is given by

$$C_{BS-EC}(n;t)=\frac{n(t+1)}{3}-1+\gamma t.$$

Similarly, we have the following performance formula for Algorithm **WS-EC**:

$$C_{WS-EC}(n;t,w)=\left(\frac{t+1}{w+2}+2^{w-1}+\gamma-1\right)n$$
$$+\gamma(t-w)-1$$

The storage requirement and optimal window sizes for Algorithm **WS-EC** are the same as those given in Theorem 3.

We can also improve Algorithm **BG** using signed encoding. Suppose that each multiplier c_i is represented in base 2^w as in Sect.III.4. Since a w-bit number $c_{i,j}$ can always be encoded into an integer $\widetilde{c_{i,j}}$ whose absolute value is less than or equal to 2^{w-1}, we can encode the entire c_i as

$$c_i=\sum_{j=0}^{h-1}\widetilde{c_{i,j}}2^{jw},$$

where $h = \lceil \frac{t+1}{w} \rceil$ and $0 \le |\widetilde{c_{i,j}}| \le 2^{w-1}$. We can easily derive the expected number of elliptic additions required by Algorithm **BG-EC** from the corresponding fornula in Sect.III.4 :

$$\begin{aligned} C_{BG-EC}(n;t,w) = & \left(\frac{2^w-1}{2^w}h + \frac{2^r-1}{2^r}\right)n \\ & + h'(2^{w-1}-1) + 2^{w'} - 2 \\ & + r(t+1-w+vw(n-1)), \end{aligned}$$

where $h = \lfloor \frac{t+1}{w} \rfloor$, $r = t+1 \mod w$, $h' = \lfloor \frac{h}{v+1} \rfloor$, $0 \le v < h$, and $w' = r$ if $h = 0 \mod v+1$ and $w' = w-1$ otherwise. Given t and v, we can find optimal window sizes depending on n as before.

4.2 Further Improvement Using Simultaneous Inversion

Another speedup technique in elliptic curve arithmetic is to take advantage of parallel computation. Suppose that it is allowed to add or double m elliptic points in parallel and suppose that we do the arithmetic in affine coordinates. Then we can use the simultaneous inversion trick due to Montgomery [9, Algorithm 10.3.4] to reduce the number of inversions required for m elliptic additions/doublings to only one at the cost of 3 field multiplications per elliptic addition/doubling. Using this technique, we can achieve about 20 to 30% speedup in Algorithms **LL-EC** and **BG-EC**, thanks to the very large degree of parallelism in these algorithms. For example, on-line precomputation $2^w Y_i$ $(0 \le i < n)$ in Algorithm **BG-EC** can be performed in affine coordinates only using one inversion, $(5n-3)$ multiplications and $2n$ squarings. Thus, if the degree of parallelism (n in the above example) is quite large, we can perform an elliptic addition only using about 5 multiplications and 2 squarings. This is the fastest known method for elliptic addition.

Note that elliptic curve arithmetic in projective coordinates usually yields better performances than elliptic curve arithmetic in affine coordinates. However, if it is possible to do parallel execution of a number of elliptic additions/doublings as discussed above, it is almost always more efficient to do elliptic curve arithmetic in affine coordinates.

Ⅵ. Batch Verification of Digital Signatures

There are a number of cryptographic applications requiring multi-exponentiation: e.g., (batch) verification of ElGamal-type signatures, elliptic scalar multiplication using Frobenius expansion, exponentiation in GF(p^n), etc. Batch verification of ElGamal-type signatures is particularly interesting, since we can substantially improve the performance of a variety of applications using digital signatures. There have been proposed and analyzed efficient batch verification algorithms on a variant of DSA signatures [5,6]. But there is no algorithm presented for efficient evaluation of multi-exponentiation required for the batch test.

A DSA signature on a message m, generated by a secret/public key pair $(x, y = g^x \bmod p)$, consists of (r, s) computed by $\lambda = g^k \bmod p$ $(k \in Z_q)$, $r = \lambda \bmod q$ and $s = k^{-1}(m + rx) \bmod q$ (p, q primes s.t. $q | p-1$ and $l = |q| = 160$, g an element of order q). Verification of the signature can be done by checking the equality $r = (g^a y^b \bmod p) \bmod q$, where $a = ms^{-1} \bmod q$ and $b = rs^{-1} \bmod q$. Since a batch verification technique cannot be applied to DSA signatures in their original form, Naccache et al. [5] considered a slight modification: send (λ, s), instead of (r, s), as a signature and convert the signature back to its original form after successful verification.

Now, let us consider batch verification of the above modified DSA signatures. First, consider n signatures, $\{(\lambda_i, s_i)\}$ $(0 \le i < n)$, generated by the same signer with a signing key pair (x, y). For convenience, we will denote the batch instance by $\{(\lambda_i, a_i, b_i)\}$ $(0 \le i < n)$, where $a_i = m_i s_i^{-1} \bmod q$ and $b_i = r_i s_i^{-1} \bmod q$. Then the batch verification equation is given by

$$\prod_{i=0}^{n-1} \lambda_i^{c_i} = g^a y^b \bmod p, \qquad (8)$$

where $a = \sum_{i=0}^{n-1} c_i a_i \bmod q$, $b = \sum_{i=0}^{n-1} c_i b_i \bmod q$ and c_i's are integers randomly chosen over the interval $[0, 2^t)$. It is proved in [5,6] that the error probability of this test is less than 2^{-t}. So, $t = 30 \sim 60$ would be sufficient in most applications. The left-hand side of equation (8) can now be efficiently computed using one of the algorithms presented in Sect.III according to the batch size n. Note that the bucket test in [6] can significantly improve the above small exponent test, in particular for large n. But our multi-exponentiation algorithms can further speed up the computation required by the bucket test.

On the other hand, it is much more likely in most applications of digital signatures that a batch instance consists of digital signatures from different signers. So, we next consider a batch instance consisting of n signatures, $\{(\lambda_i, a_i, b_i)\}$ $(0 \le i < n)$, generated by n distinct signers, where each signer i possesses a signing key pair (x_i, y_i). In this case, the batch verification equation becomes

$$\prod_{i=0}^{n-1} \lambda_i^{c_i} = g^a \prod_{i=0}^{n-1} y_i^{b_i'} \bmod p, \qquad (9)$$

where $a = \sum_{i=0}^{n-1} c_i a_i \bmod q$, $b_i' = c_i b_i \bmod q$. Now, since each b_i' is $l = |q| = 160$ bits long, the main computational load is to evaluate multi-exponentiation in the right-hand side of equation (9) and the bucket test in [6] does not result in any improvement in this case. Thus, a fast multi-exponentiation algorithm is crucial to the practicality of this general batch verification. Equation (9) can be rewritten for more efficient computation as

$$\prod_{i=0}^{n-1} y_i^{b_i'} \lambda_i^{c_i} = g^a \bmod p, \qquad (10)$$

where $b_i' = -c_i b_i \bmod q$.

The left-hand side of equation (10) can be computed using one of our presented algorithms according to the batch size n. Since the size of the c_i is much smaller than that of the b_i', we may choose different window sizes in Algorithms **WU**, **WS** and **LL** for better performances. However, Algorithm **BG** should have the same window size to share some common computations. From the analysis of Sect.IV, we can see that the presented algorithms enable us to verify a batch instance of n signatures about 2 to 4 times faster than the naive binary algorithm, and about 2 to 10 times faster than individual verification, depending on the batch size n.

Ⅶ. Conclusion

There are a number of cryptographic applications requiring efficient multi-exponentiation(i.e., computation of a product of powers), in particular in a variety of applications using digital signatures. In this paper, we presented several algorithms for efficient multi-exponentiation and analyzed their performances. The presented algorithms can perform n-term exponentiation(n ranging from a few to several thousands) 2 to 4 times faster than the basic binary multi-exponentiation algorithm. We can choose the best performing algorithm according to the number (n) of powers and the bit-length of exponents. In general, Algorithm **WU** is the best for very small n, while for moderate values of n Algorithms **LL** and **WS** perform better. For very large n, Algorithm **BG** is the fastest.

참 고 문 헌

[1] J. Bos and M.Coster, "Addition chain heuristics," *Advances in Cryptology - Crypto'89*, LNCS 435, Springer-Verlag, 1990, pp. 400~407.

[2] H. Cohen, A. Miyaji and T. Ono, "Efficient elliptic curve exponentiation," *Information and Communications Security*, LNCS 1334, S.V., 1997, pp. 282~290.

[3] E. F. Brickell, D. M. Gordon, K. S. McCurley and D. Wilson, "Fast exponentiation with precomputation," *Advances in Cryptology-Eurocrypt'92*, LNCS 658, S.V., 1993, pp. 200~207.

[4] C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation," *Advances in Cryptology-Crypto'94*, LNCS 839, S.V., 1994, pp. 95~107.

[5] D. Naccache, D. M' Raihi, S. Vaudenay and D.Raphaeli, "Can D.S.A. be improved? Complexity trade-offs with the digital signature standard," *Advances in Cryptology-Eurocrypt'94*, LNCS 950, S.V., 1994, pp. 77~85.

[6] M. Bellare, J. A. Garay and T. Rabin, "Fast batch verification of modular exponentiation and digital signatures," *Advances in Cryptology-Eurocrypt'98*, LNCS 1403, S.V., 1998, pp. 236~250.

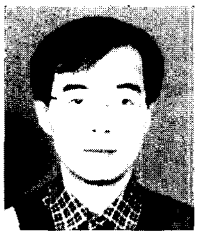
[7] S. M. Yen, C. S. Laih and A. K. Lenstra, "Multi-exponentiation," *IEE Proc. of Computers and Digital Techniques*, Vol. 141, 1994, pp. 325~326.

[8] S. G. Sim and P. J. Lee, "An efficient implementation of two-term exponentiation in elliptic curves," *Proc. of Japan-Korea Joint Workshop on Information Security and Cryptology(JW-ISC 2000)*, Jan. 25-26, 2000, Naha, Okinawa, Japan, pp. 61~68.

[9] H. Cohen, *A course in computational number theory*, GTM 138, S.V., 1993, Third corrected printing, 1996.

[10] J. A. Solinas, "An improved algorithm for arithmetic on a family of elliptic curves," *Advances in Cryptology - Crypto'97*, LNCS 1294, S.V., 1997, pp. 357~371.

〈著者紹介〉



임 채 훈 (Chae Hoon Lim) 정회원
1989년 2월 : 서울대학교 전자공학과 졸업
1992년 2월 : 포항공과대학교 전자전기공학과 석사
1996년 2월 : 포항공과대학교 전자전기공학과 박사
1996년 3월~2002년 2월 : (주)퓨처시스템 암호체계센터
2002년 3월~현재 : 세종대학교 인터넷학과 조교수
〈관심분야〉 암호 알고리즘/프로토콜 설계/분석, 인터넷 보안